Persistent

Core Java: JDBC II

Persistent Interactive | Persistent University

Objectives :

- JDBC Architecture
- JDBC Drivers
- JDBC API
- Transactions

What is JDBC

- It is not an acronym, but is called Java Database Connectivity
- It is a vendor independent API drafted by Sun to access data from different databases in a consistent and reliable way
- JDBC provides an API by hiding the vendor specific API by introducing the concept of a JDBC driver between the application and the database API
- Hence, JDBC requires a vendor specific driver
- The JDBC driver converts the JDBC API calls from the Java application to the vendor specific API calls

Main goals of JDBC

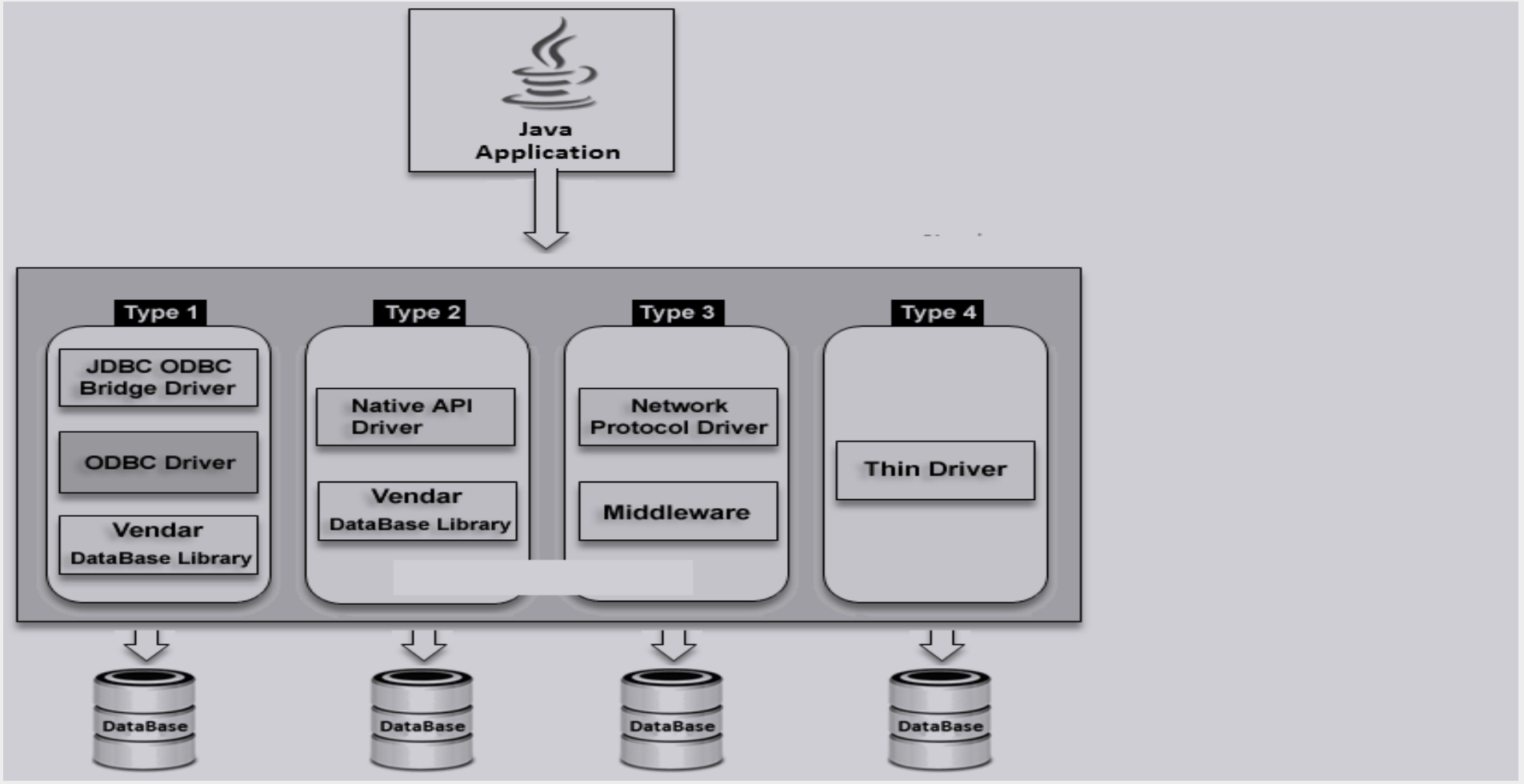
- JDBC should be an SQL level API
- JDBC should capitalize on the experience of the existing database APIs
- JDBC should provide a simple programming interface

JDBC Driver Types

There are 4 types of JDBC drivers

- Type 1 – JDBC – ODBC Bridge
- Type 2 – Native API – Partly Java Driver
- Type 3 – Java – Net Protocol Driver
- Type 4 – 100% Java Driver

JDBC Driver Types

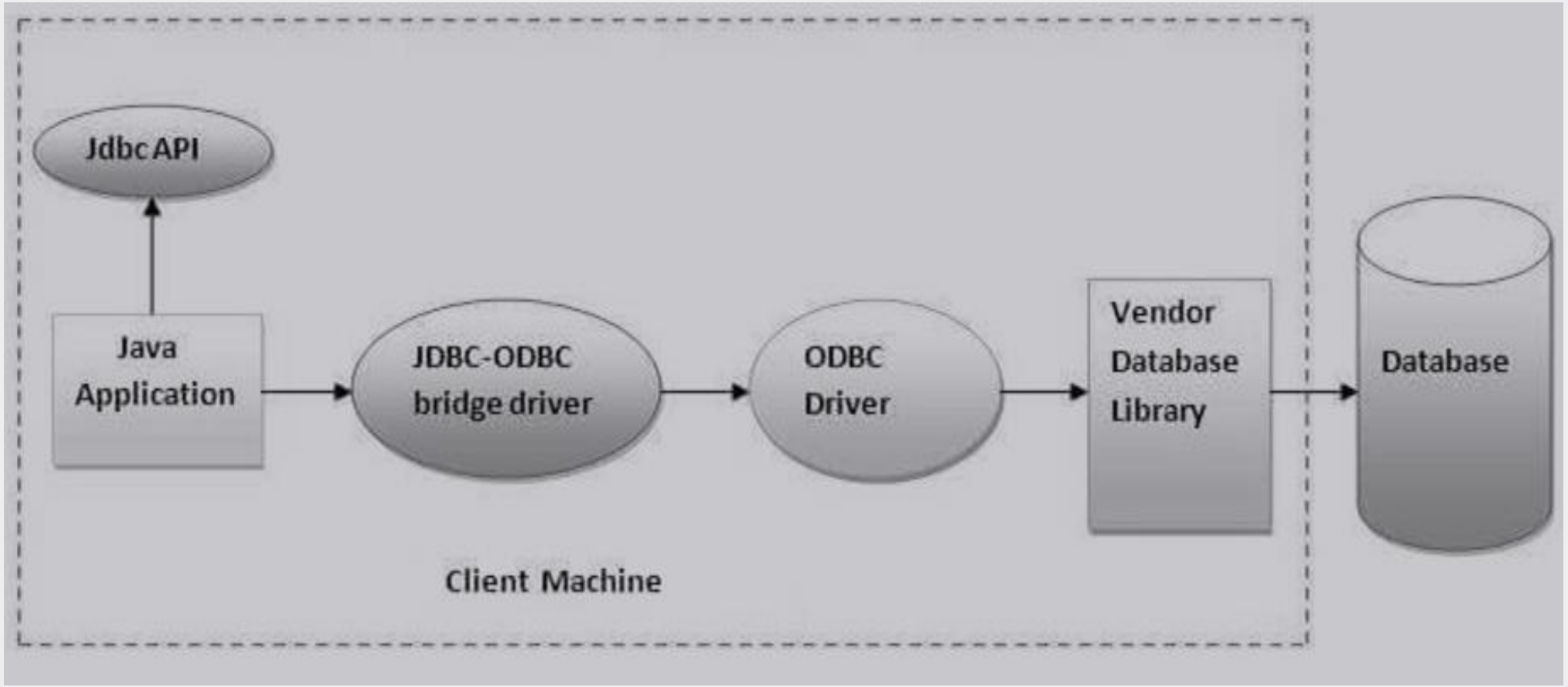


Persistent

JDBC Type 1 Driver

- Translates all JDBC calls to ODBC (Open Database Connectivity) calls and sends them to the ODBC driver
- This requires the ODBC driver to be present in the client’s machine
- Two stages of data type conversions occurs

JDBC Type 1 Driver

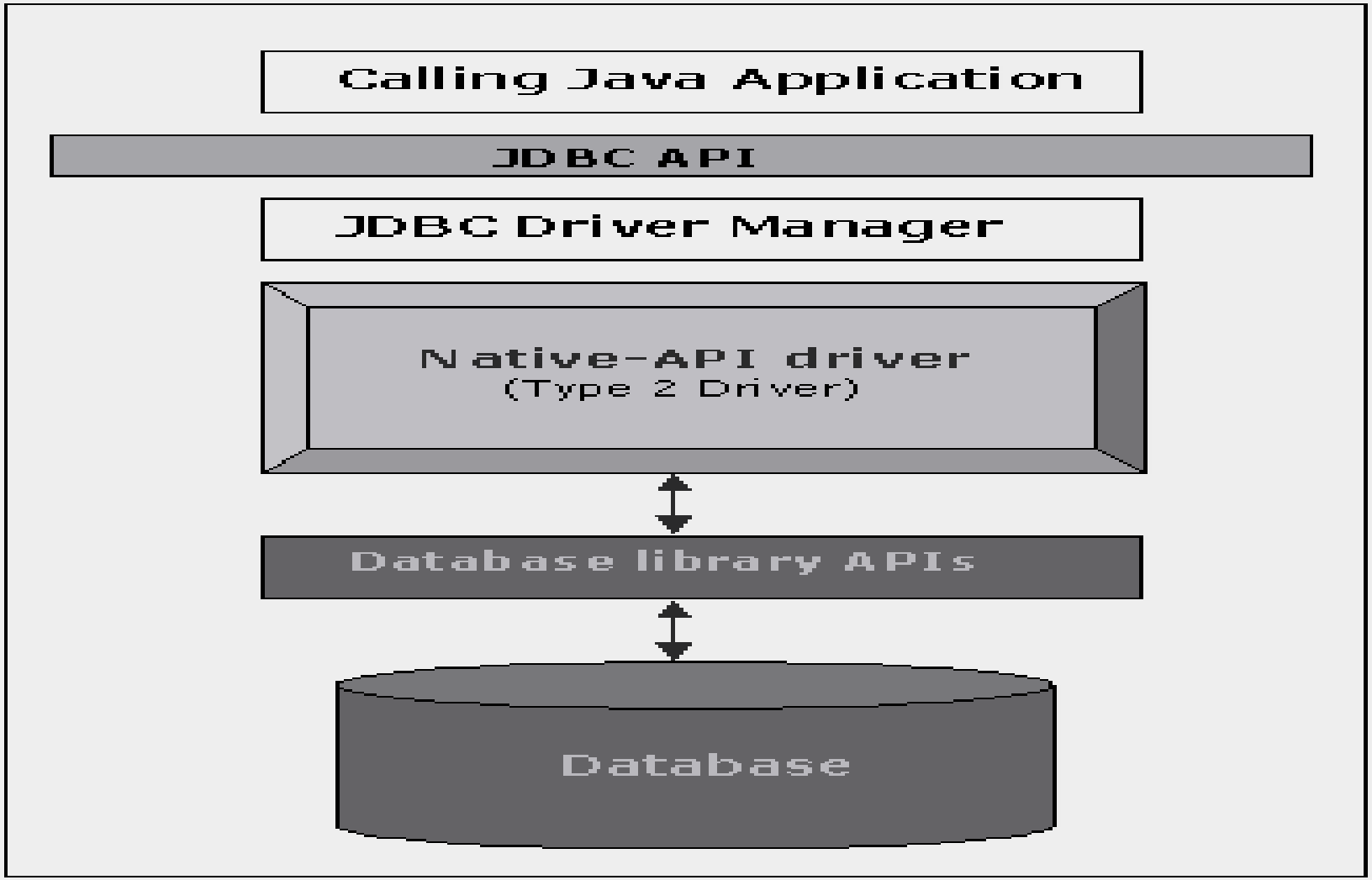


Persistent

JDBC Type 2 Driver

- Converts JDBC calls into database specific calls for databases such as Oracle, SQL Server etc.
- Communicates directly with the database server
- Requires some binary code modules to be present on the client machine
- More efficient than JDBC – ODBC bridge as there are fewer layers of communication and translation

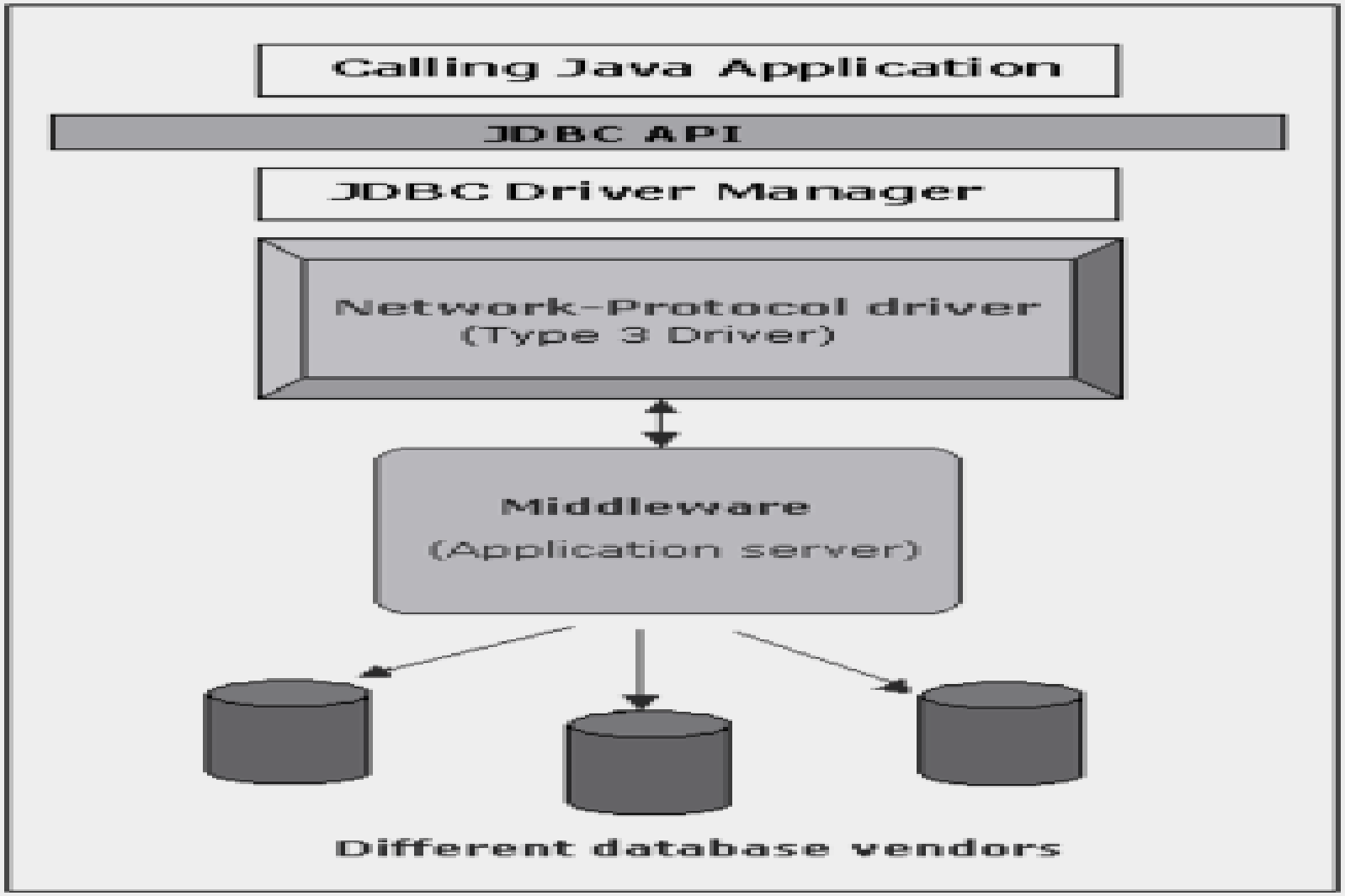
JDBC Type 2 Driver



JDBC Type 3 Driver

- Is a pure Java driver
- Follows a three tiered approach
- JDBC calls are passed through the network to the middle tier server
- The middle tier server then translates the requests to database specific native connectivity interface to further the request to the database server

JDBC Type 3 Driver

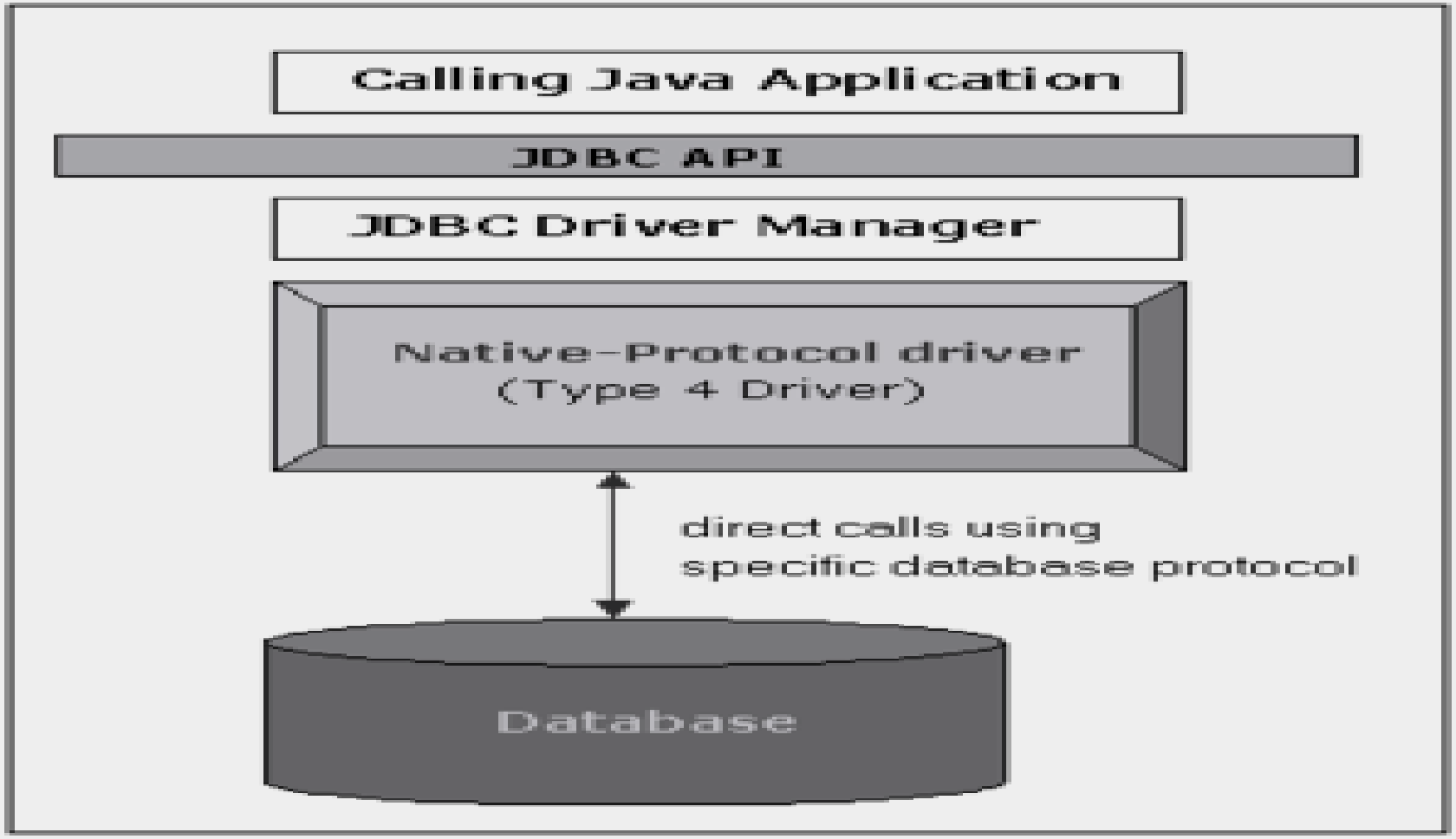


Persistent

JDBC Type 4 Driver

- Is a pure Java driver
- Converts JDBC calls to vendor specific database management system protocol
- Hence client applications can directly communicate with the database server
- Completely platform independent, hence are free of deployment and management issues

JDBC Type 4 Driver



Configuring JDBC Drivers

- You just need to make the JDBC driver classes available to the program you are writing

To do this, there is only one step:

- Add the JDBC driver jar to CLASSPATH, or
- Use the –classpath option to make the JDBC driver classes available, or
- Add the JDBC driver jar file to the jre/ext directory

JDBC Objects – A recap

- The basic interfaces in the java.sql package are
  - Connection
  - Statement
  - ResultSet

PreparedStatement : Pre-compiled Queries

- The basic concept
  - An SQL query when executed against a database undergo a parsing and compilation stage
  - Once a query has been compiled, a query plan is created. These two stages are resource intensive
  - If similar queries are fired multiple times, there is no need for the compilation and parsing to happen multiple times
  - In such scenarios, PreparedStatements are very efficient

PreparedStatement : Pre-compiled Queries

- The basic usage
  - Create a statement in standard form that is sent to the database for compilation before actually being used
  - Each time you use it, simply replace some of the marked parameters using the setXXX () methods

- PreparedStatement is an interface that inherits from Statement. The following methods are inherited
  - execute ()
  - executeQuery ()
  - executeUpdate ()

PreparedStatment Example

```
try {

    //Assume, the url, userName, and password strings
    have been initialized

    Connection
    connection=DriverManager.getConnection (url,
    userName, password);

    String sqlQuery = “UPDATE EMPLOYEE SET
    SALARY = ? WHERE EMPID = ?”;
```

PreparedStatment Example

```
PreparedStatement ps = connection.prepareStatement
    (sqlQuery);

    int[] newSalaries = getSalariesForAllEmployees();
    int[] employeeIDs = getAllIds ();

    for (int i = 0; i < employeeIDs.length; i++) {
        ps.setInt (1, newSalaries[i]);
        ps.setInt (2, employeeIDs[i]);
        ps.executeUpdate ();
    }}

    catch (SQLException sqle) {
    System.out.println ("Error while executing prepared
    statement");   }
```

PreparedStatment Methods

- **setXxx ()**
  - Sets the indicated parameter (?) in the SQL statement to the passed in value

- **clearParameters**
  - Clears all set parameter values in the statement

CallableStatment

- Stored procedures can greatly improve performance since they are pre-compiled units lying on the database server
- Usually involves making a choice between splitting business logic between the Java program and the database code
- Used to invoke stored procedures

CallableStatment Example

```
CallableStatement cstmt = conn.prepareCall("{?=call
GETBOOKID(?)}");

cstmt.registerOutParameter(1, java.sql.Types.INTEGER);
                    cstmt.setString(2, "The Big Sleep");
                    cstmt.execute();

}

catch (SQLException ex)      {…}
```

Metadata from DB

- A Connection's database is able to provide schema information describing its tables, its supported SQL grammar, its stored procedures the capabilities of this connection, and so on
- This information is made available through a DatabaseMetaData object.

Metadata from DB - example

```
Connection con = …. ;

DatabaseMetaData dbmd = con.getMetaData();

String catalog = null;
String schema = null;
String table = “sys%”;
String[ ] types = null;

ResultSet rs =
   dbmd.getTables(catalog , schema , table , types );
…
```

Metadata from DB - example

```
try {//Assume that connection “conn” has already been
        //created
        DatabaseMetaData metaData =
  conn.getMetaData ();

        String databaseProductName =
        metaData.getDatabaseProductName ();
        String databaseVersion =
        metaData.getDatabaseProductVersion ();
        }

        catch (SQLException ex) {
        System.out.println (“Exception while retrieving
  meta-data”);
        }
```

JDBC – Metadata from ResultSet

```
public static void printRS(ResultSet rs) throws
    SQLException
{
    ResultSetMetaData md = rs.getMetaData();
    // get number of columns
    int nCols = md.getColumnCount();

// print column names
    for(int i=1; i < nCols; ++i)
                System.out.print( md.getColumnName( i)+",");
/ / output resultset
    while ( rs.next() )
    {           for(int i=1; i < nCols; ++i)
                System.out.print( rs.getString( i)+",");
                System.out.println( rs.getString(nCols) );
     }}
```

JDBC 2.0 Enhancements

- ResultSet navigation
- Batch Updates
- get Blob, get Clob
- getObject

ResultSet Navigation

New ResultSet Operations

first(), last(), next()

previous(), beforeFirst(), afterLast()

absolute(int), relative(int)

Rows may be updated and inserted

rs.update(3, “new filename”); rs.updateRow();

Rows may be deleted

Batch Updates

```
con.setAutoCommit(false);

Statement s  = con.createStatement();
s.addBatch(….);
s.addBatch(…..);
……
s.executeBatch();

con.commit();
```

JDBC 2 – Scrollable Result Set

```
Statement stmt =
con.createStatement(ResultSet.TYPE_SCROLL_INSENSI
    TIVE,   ResultSet.CONCUR_READ_ONLY);

String query = “select students from class where
    type=‘not sleeping’ “;

ResultSet rs = stmt.executeQuery( query );

rs.previous();  / /go back in the RS (not possible in JDBC
    1…)

rs.relative(-5); / / go 5 records back
rs.relative(7); / / go 7 records forward
rs.absolute(100); / / go to 100th record
```

JDBC 2 – Updateable ResultSet

```
Statement stmt =
con.createStatement(ResultSet.TYPE_FORWARD_ONLY,
ResultSet.CONCUR_UPDATABLE);

String query = " select students, grade from class
    where type='really listening this presentation☺' '';

ResultSet rs = stmt.executeQuery( query );

while ( rs.next() )
{
    int grade = rs.getInt("grade");
    rs.updateInt("grade", grade+10);
    rs.updateRow();
}
```

RowSet Interface

- Interface RowSet
  - Configures the database connection automatically
  - Prepares query statements automatically
  - Provides set methods to specify the properties needed to establish a connection
    - Part of the javax.sql package
  - Two types of RowSet
  - Connected RowSet
  - Connects to database once and remain connected
    - Disconnected RowSet
- Connects to database, executes a query and then closes connection

RowSet Interface

- Package javax.sql.rowset
  - JdbcRowSet
  - Connected RowSet
  - Wrapper around a ResultSet
  - Scrollable and updatable by default
  - CachedRowSet
  - Disconnected RowSet
  - Cache the data of ResultSet in memory
  - Scrollable and updatable by default
  - Serializable

RowSet interface

- Limitation
  - Amount of data that can be stored in memory is limited

Some Problem when using ResultSet

- Insert or Delete or Update on some database is not working properly.
- We can solve this problem by using RowSet.

Insert data by using RowSet(1)

```
package nasora;

import java.sql.*;
import javax.sql.rowset.*;
import oracle.jdbc.rowset.*;

public class Example6 {
   public static void main(String[] args){
       try {
           String sql = "select * from books";

           OracleCachedRowSet crset = new
    OracleCachedRowSet();

    crset.setUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl");

    crset.setUsername("nas");

    crset.setPassword("nas123");

    crset.setCommand(sql);

    crset.execute();
```

Insert data by using RowSet(2)

```
crset.setReadOnly(false);

crset.moveToInsertRow();

crset.updateInt(1, 5);

crset.updateString(2, "Thinking");

crset.updateString(3, "Nirata");

crset.insertRow();

crset.acceptChanges();

crset.close();}

 catch(SQLException e){

    System.out.println("Error : "+e.getMessage());
   }
} }
```

Update Data by using RowSet

```
package nasora;

import java.sql.*;
import oracle.jdbc.rowset.*;

public class Example7 {
   public static void main(String[] args){
       try {
           String sql = "select * from books";

          OracleCachedRowSet crset = new
    OracleCachedRowSet();

       crset.setUrl("jdbc:oracle:thin:@127.0.0.1:1521:orcl");

       crset.setUsername("nas");

       crset.setPassword("nas123");

       crset.setCommand(sql);
```

Update Data by using RowSet

```
    crset.execute();

    crset.setReadOnly(false);

     crset.absolute(2);

    crset.updateInt(1, 2);
     crset.updateString(2, "Work Smart");
     crset.updateString(3, "Amorn");

    crset.updateRow();

    crset.acceptChanges();
    crset.close();

}
catch(SQLException e){
            System.out.println("Error : "+e.getMessage;
        }} }
```

Transactions

- By default, after each SQL statement is executed, the changes are automatically committed to the database
- Sometimes you handle a task/a group of tasks that need to be completed in an assured manner for any meaningful operation to be successful
- Turn auto-commit off to group two or more statements together in a transaction
- connetion.setAutoCommit (false);
- Calling commit () permanently records the changes in the databse
- Call rollback () if any error occurs

Transaction Control Example

```
package nasora;
import java.sql.*;

public class Example8 {
   public static void main(String[] args){

    Connection conn = null;
     try {

     Class.forName("oracle.jdbc.OracleDriver");

    conn = DriverManager.getConnection("jdbc:oracle:thin:
                @127.0.0.1:1521:orcl", "nas", "nas123”);

    String sql = "insert into books(id, title, author)values(?,
    ?, ?)”;

     conn.setAutoCommit(false);

    PreparedStatement pstmt =conn.prepareStatement(sql);

     pstmt.setInt(1, 2);
```

Transaction Control Example

```
            pstmt.setString(2, "TO BE ONE");
              pstmt.setString(3, "Nicky");
              int retVal = pstmt.executeUpdate();
              conn.commit();
              conn.setAutoCommit(true);
              pstmt.close();
              conn.close();
    }
catch(ClassNotFoundException e){
              try{
                        conn.rollback();
              }
              catch(SQLException ex){}
              System.out.println("Error : "+e.getMessage());
      }
```

Transaction Control Example

```
    catch(SQLException e){

    try{
                        conn.rollback();
        }

    catch(SQLException ex){}

            System.out.println("Error : "+e.getMessage());
    }
finally{

            if(conn!=null)
            try
            {
            conn.setAutoCommit(true);
            conn.close();
            }
            catch(SQLException e){}
            }   }   }
```

Connecting with DataSource

- DataSource Interface is one of the preferred means of getting a connection to a data source.
- It provide connection pooling and distributed transactions.
- It provides loose coupling for connectivity results ease in switching databases.
- Modern JDBC drivers provide implementations of ConnectionPoolDataSource and PooledConnection
- It makes possible to build a much smaller connection pool manager.

How to create DataSource ?

- Code for Obtaining MySQL connection with Data source

```
Class.forName("com.mysql.jdbc.Driver");
MysqlConnectionPoolDataSource dataSource=new MysqlConnectionPoolDataSource();
 dataSource.setDatabaseName("test");
dataSource.setServerName("localhost");
dataSource.setPort(Integer.parseInt("3306"))
dataSource.setUser("root");

dataSource.setPassword("root");

PooledConnection cn = dataSource.getPooledConnection();
```

Connection Pooling

- A connection pool is a cache of database connection objects.
- It is a technique of creating and managing a pool of connections that are ready to use.
- Connection pools promote the reuse of connection objects and reduce the number of times that connection objects are created.
- It significantly improves performance for database-intensive applications as creating connection objects is costly in terms of time and resources.

Benefits of Connection Pooling

- Reduced connection creation time
- Simplified programming model
- Controlled resource usage

New features in JDBC 4.2

- Addition of REF_CURSOR support.
- Addition of java.sql.DriverAction Interface
- Addition of the java.sql.SQLType Interface
- This interface is used to create an object that identifies a generic SQL type, called JDBC type or a vendor specific data type.

New features in JDBC 4.2

- Addition of the java.sql.JDBCType Enum
- This enum identifies generic SQL Types, called JDBC Types. Use JDBCType in place of the constants defined in Types.java.
- Add Support for large update counts
- return a long value for the update count.
- Changes to the existing interfaces
- Rowset 1.2: Lists the enhancements for JDBC RowSet.

Summary

With this we have come to an end of our session, where we discussed about

- JDBC basics
- Types of drivers
- PreparedStatement & CallableStatement, ResultSet, RowSet interfaces
- Transactions
- Batch updates
- Connection pooling
- JDBC 4.2 features

Appendix

References

Thank you

Reference Material : Websites & Blogs

- https://docs.oracle.com/javase/8/docs/technotes/guides/jdbc/jdbc_42.html
- https://docs.oracle.com/javase/8/docs/api/java/sql/DriverAction.html
- https://docs.oracle.com/javase/8/docs/api/java/sql/SQLType.html
- https://docs.oracle.com/javase/8/docs/api/java/sql/JDBCType.html

Reference Material : Books

- **Head First Java**
  - By: Kathy Sierra, Bert Bates
  - Publisher: O’Reilly Media, Inc.
- **Java Complete Reference**
  - By Herbert Schildt

Persistent

Thank you!

Persistent Interactive | Persistent University